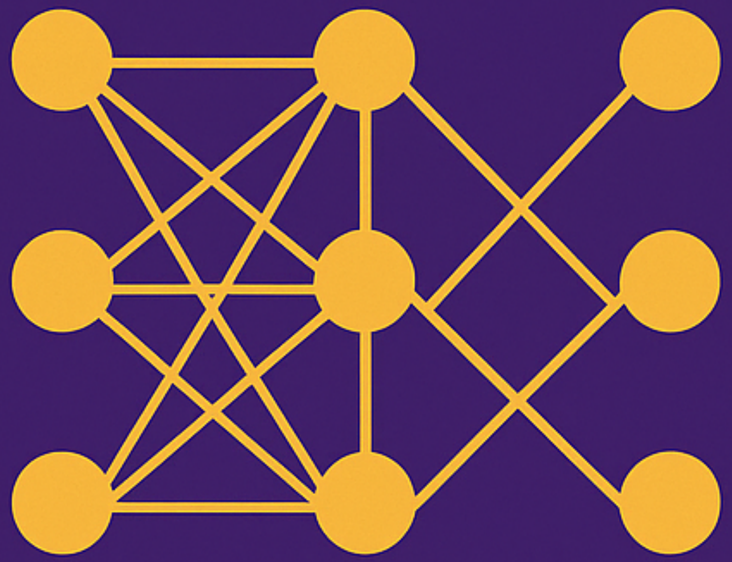
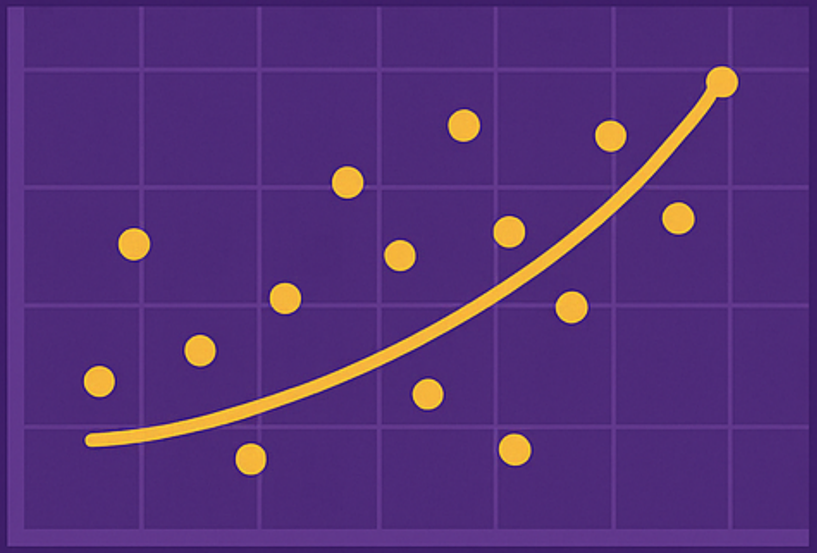
Modern CNNs

Overview of the main CNN algorithms up to these days



INTRODUCTION TO MACHINE LEARNING



FALL 2025 LSU

LeNet

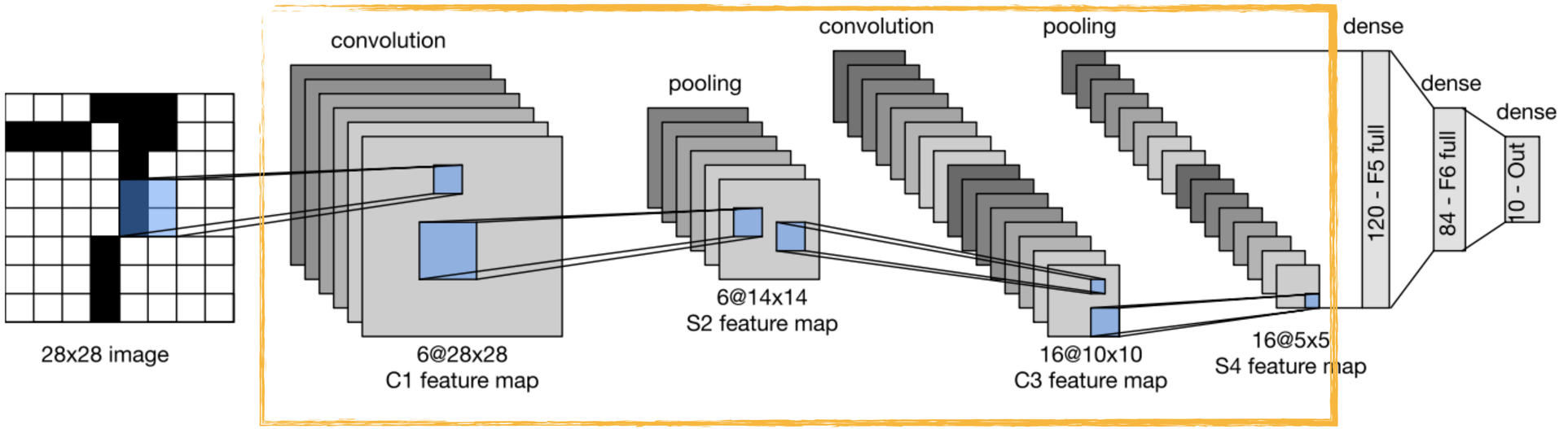
We now have all the ingredients required to assemble a fully-functional CNN

Introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images:

- LeCun's team published the first study to successfully train CNNs via backpropagation (LeCun *et al.*, 1989)
- *LeNet is* among the first published CNNs (LeCun *et al.*, 1998)
- outstanding results:
  - matching the performance of support vector machines,
  - became a dominant approach in supervised learning,
  - achieved an error rate of less <1%

To this day, some ATMs still run the code that Yann LeCun and his colleague Leon Bottou wrote in the 1990s!

LeNet



2 convolutional blocks, in each:

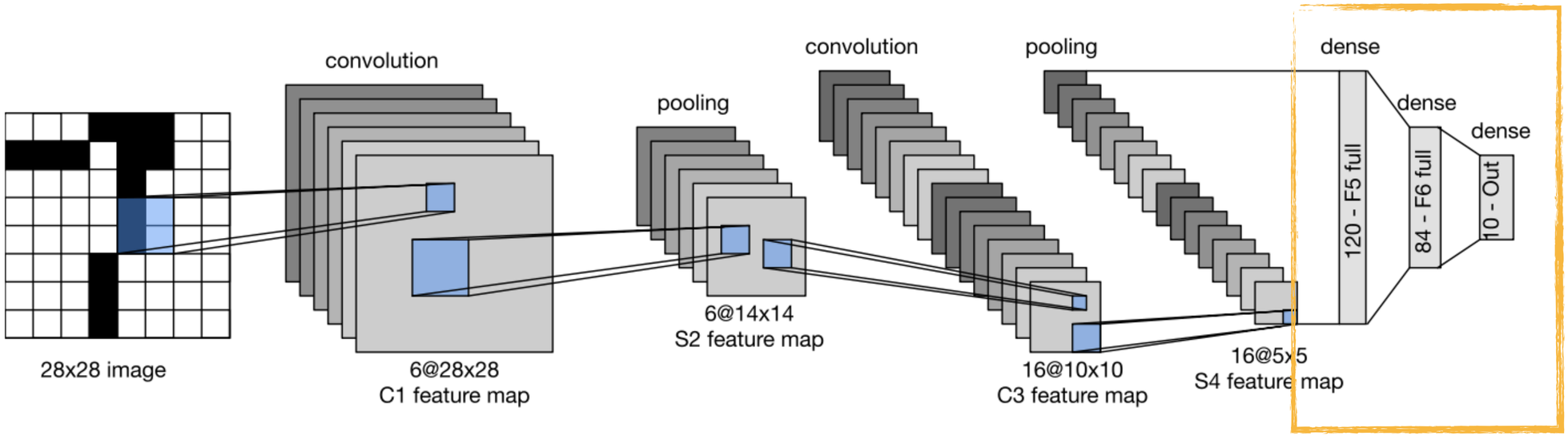
- a convolutional layer (5 × 5 kernel),
- a sigmoid activation function,
- average pooling operation.

In each pooling layer:

- 2 × 2 pooling operation (stride 2)
- reduces dimensionality by x4 (spatial downsampling.)

- **Note 1**: increasing the number of channels in each convolutional block (6 –>16)
- **Note 2**: ReLUs and max-pooling work better, they had not yet been discovered)

LeNet



Dense blocks:

- Flatten the output of the last pooling layer:
  - From 4th order tensors (b,c,h,w) —> 2nd (b, array)
  - the first dimension to index examples in the minibatch and the second to give the flat vector representation of each example.
- sequentially apply 3 fully connected (linear) layers:
  - 120d —> 84d —> 10d

Note 1: implementing such models with modern deep learning frameworks is remarkably simple:
See Notebook "LeNet"

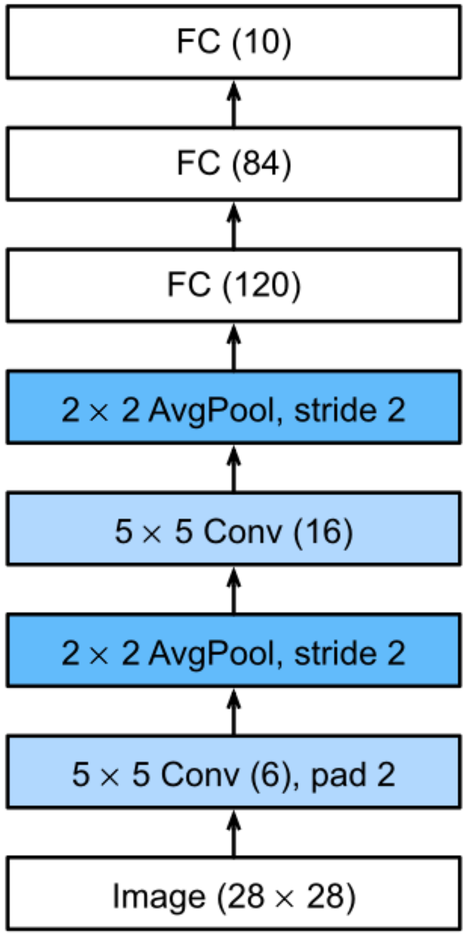
Le Net

```
def init_cnn(module):  #@save
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)

class LeNet(d2l.Classifier):  #@save
    """The LeNet-5 model."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))
```

The first convolutional layer uses two pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5 × 5 kernel.

*

Compressed notation for LeNet



* **Note**: the Gaussian activation layer was substitute by a softmax layer. This greatly simplifies the implementation, not least due to the fact that the Gaussian decoder is rarely used nowadays.

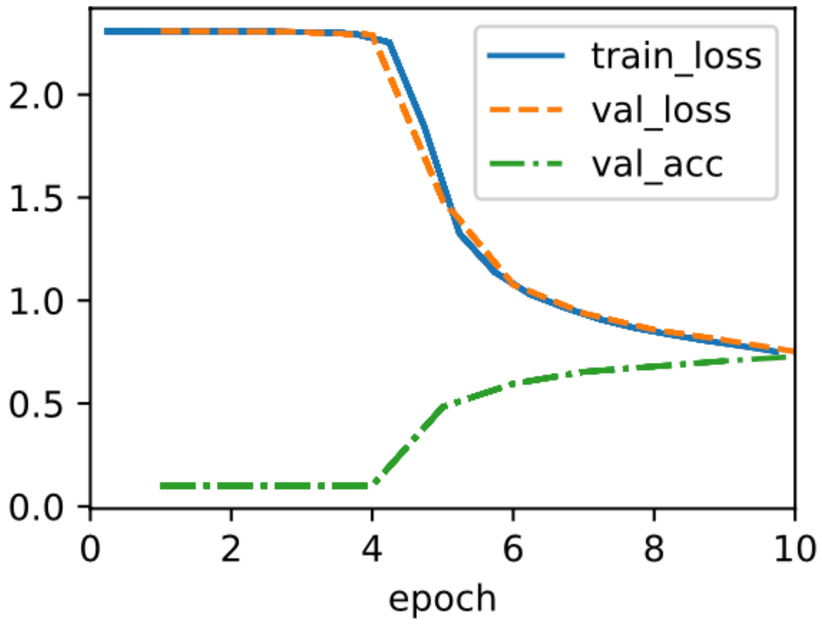
LeNet

```
@d2l.add_to_class(d2l.Classifier)  #@save
def layer_summary(self, X_shape):
    X = torch.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        print(layer.__class__.__name__, 'output shape:\t', X.shape)
model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d output shape:         torch.Size([1, 6, 28, 28])
Sigmoid output shape:        torch.Size([1, 6, 28, 28])
AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
Conv2d output shape:         torch.Size([1, 16, 10, 10])
Sigmoid output shape:        torch.Size([1, 16, 10, 10])
AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
Flatten output shape:        torch.Size([1, 400])
Linear output shape:         torch.Size([1, 120])
Sigmoid output shape:        torch.Size([1, 120])
Linear output shape:         torch.Size([1, 84])
Sigmoid output shape:        torch.Size([1, 84])
Linear output shape:         torch.Size([1, 10])
```

LeNet

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=0.1)
model.apply_init([next(iter(data.get_dataloader(True)))[0]], init_cnn)
trainer.fit(model, data)
```



Modern CNNs - a Tour

- **AlexNet** (Krizhevsky et al., 2012) — first large-scale network to beat conventional computer vision methods on a large-scale vision challenge
- **VGG** (Simonyan and Zisserman, 2014) — makes use of a number of repeating blocks of elements
- **Network in Network (NiN)** (Lin et al., 2013) — convolves whole neural networks patch-wise over inputs
- **GoogLeNet** (Szegedy et al., 2015) — uses networks with multi-branch convolutions
- **Residual Network (ResNet)** (He et al., 2016) — remains one of the most popular off-the-shelf architectures in computer vision
- **ResNeXt** (Xie et al., 2017) — introduces blocks for sparser connections
- **DenseNet** (Huang et al., 2017) — connects each layer to all subsequent layers, promoting feature reuse and alleviating vanishing gradients

Modern CNNs - a Tour

Between the early 1990s and the results of 2012, neural networks were often surpassed by other machine learning methods, e.g.:

Kernel methods (Schölkopf and Smola, 2002) — extend linear models to nonlinear problems using the kernel trick, most famously in support vector machines (SVMs).
Ensemble methods (Freund and Schapire, 1996) — combine many weak learners, such as decision trees, into a strong predictor, with AdaBoost as the canonical example.
Structured estimation (Taskar et al., 2004) — models interdependent outputs (e.g., sequences or graphs) rather than predicting each label independently, as in structured SVMs or CRFs.

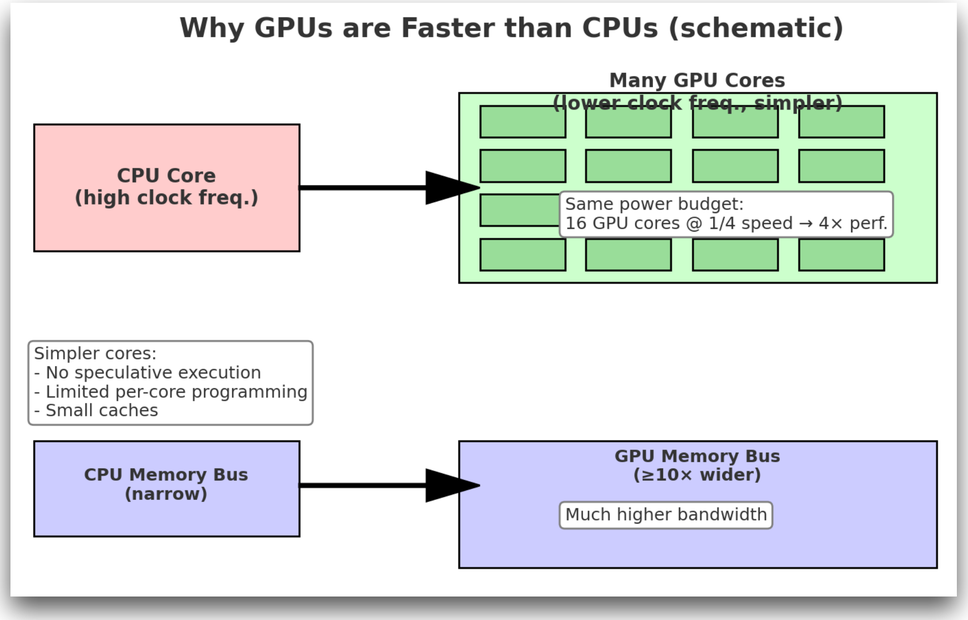
Most of the **progress** came from:

- ✳ more **clever** ideas for feature extraction and deep insight into geometry
- ✳ Exponentially increasing **computational power**
- ✳ Increased size of **data** sets
- ✳ Parameter initialization, better variants of SGD, better activation functions (non-squashing)

AlexNet

- The first modern CNN (Krizhevsky *et al.*, 2012)

- AlexNet (2012) and its precursor LeNet (1995) share many architectural elements, but
  - ImageNet dataset was released (Deng *et al.*, 2009): one million examples, 1000 each 1000 category (HD: 224 × 224)
  - Graphical processing units (GPUs): developed to accelerate graphics for video games via high-throughput 4×4 matrix–vector operations, GPUs proved well-suited for convolutional layer computations. — NVIDIA and ATI had begun optimizing GPUs for general computing operations (Fernando, 2004)



- AlexNet employed an 8-layer CNN, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a large margin

AlexNet

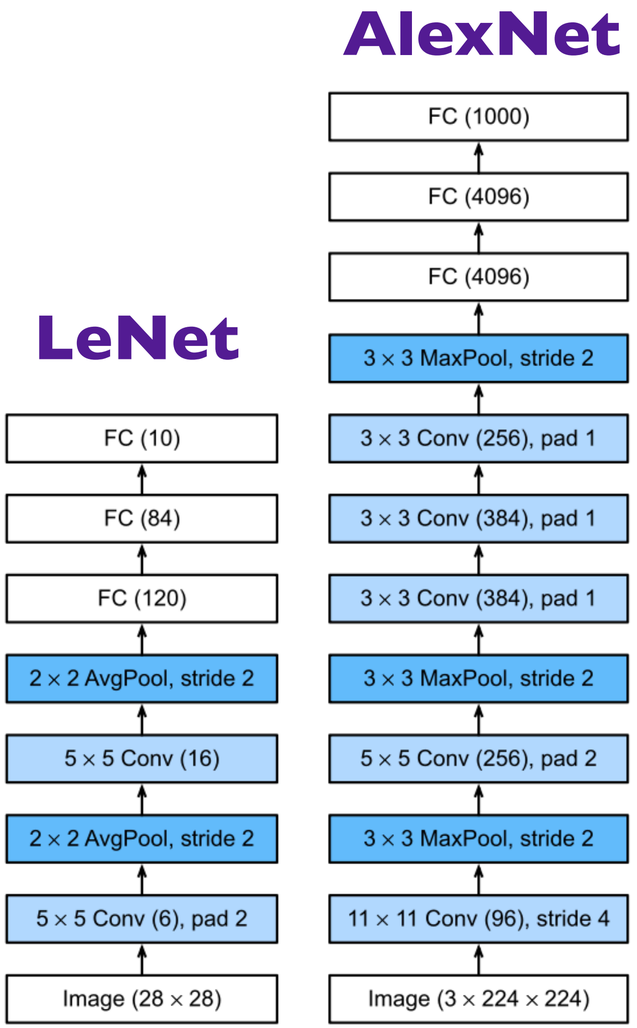
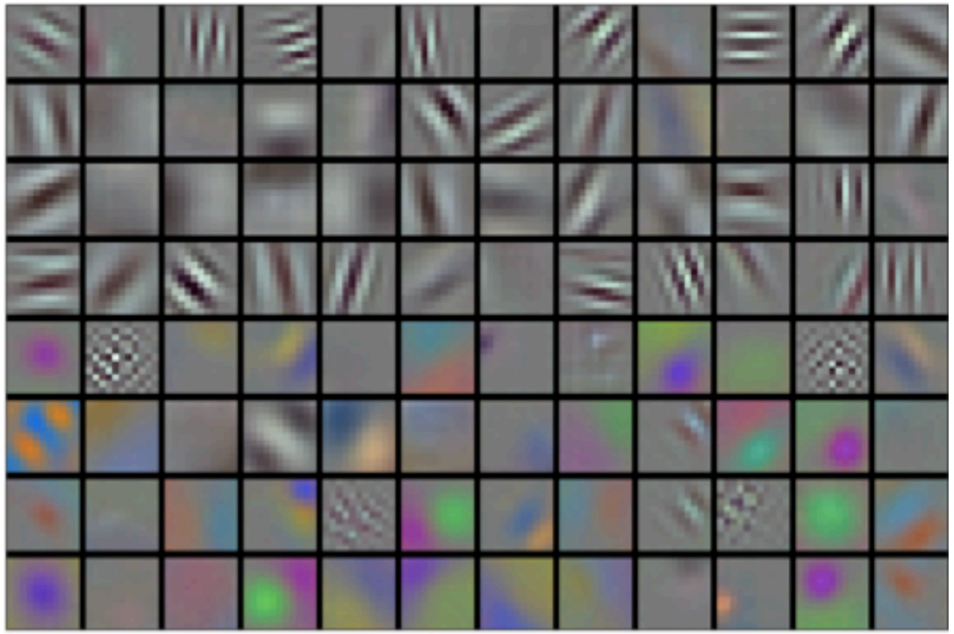


Image filters learned by the first layer of AlexNet.

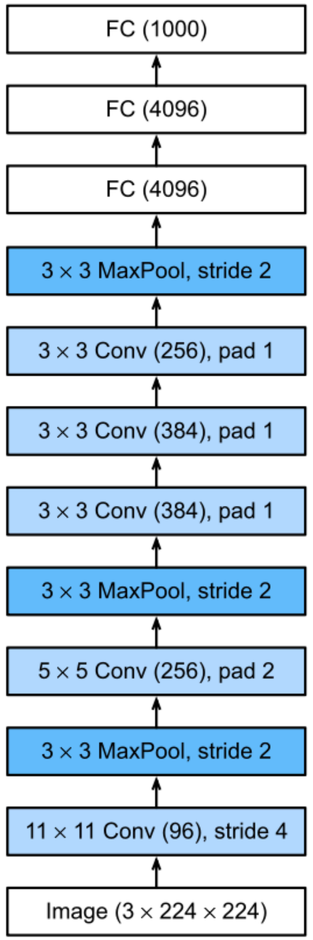


Note:

- Deeper
- Used ReLU
- 1st Conv: 11x11
- huge FC layers
  - Controlled by Dropout
- Adopted data augmentation (Buslaev *et al.* (2020))
  - flipping, clipping, and color changes.

AlexNet

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```



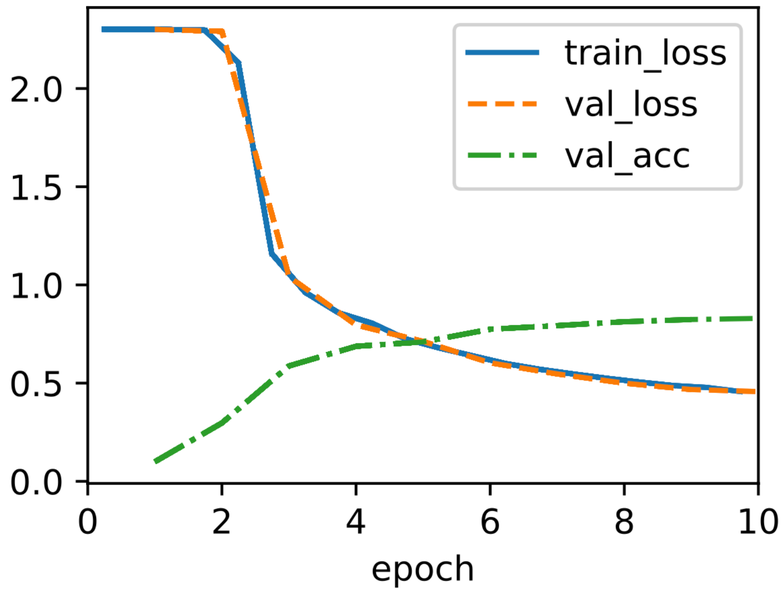
AlexNet

```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d output shape:         torch.Size([1, 96, 54, 54])
ReLU output shape:   torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Conv2d output shape:         torch.Size([1, 256, 26, 26])
ReLU output shape:   torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 256, 12, 12])
ReLU output shape:   torch.Size([1, 256, 12, 12])
MaxPool2d output shape:      torch.Size([1, 256, 5, 5])
Flatten output shape:        torch.Size([1, 6400])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

AlexNet

```
model = AlexNet(lr=0.01)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



Networks Using Blocks (VGG)

- Designing neural network architectures has grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.
- The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, in their eponymously-named *VGG* network (Simonyan and Zisserman, 2014).
  - They were primarily interested in whether deep or wide networks perform better.
  - In a rather detailed analysis they showed that deep and narrow networks significantly outperform their shallow counterparts
- The VGG block consists of a *sequence* of convolutions with 3 × 3 kernels with padding of 1 (keeping height and width) followed by a 2 × 2 max-pooling layer with stride of 2 (halving height and width after each block).

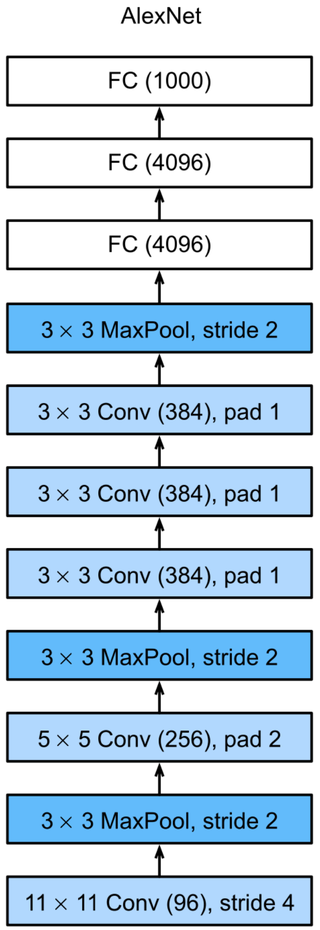
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

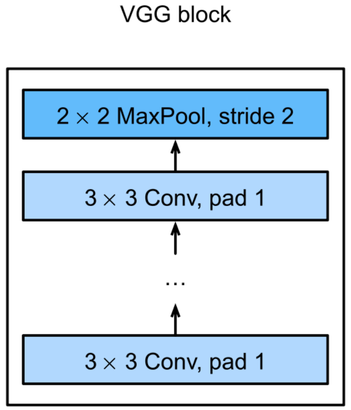


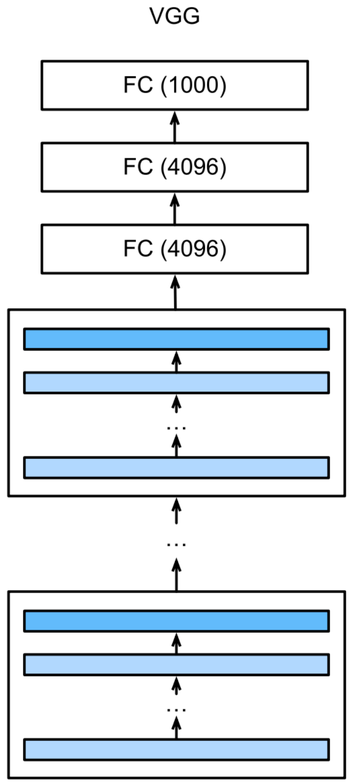

Networks Using Blocks (VGG)

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
    super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs,
                                       out_channels))
        self.net = nn.Sequential(*conv_blks, nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(),
                nn.Dropout(0.5), nn.LazyLinear(4096),
                nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

The original VGG network had 5 conv blocks, among which the first two have one conv layer each and the latter three contain two conv layers each.







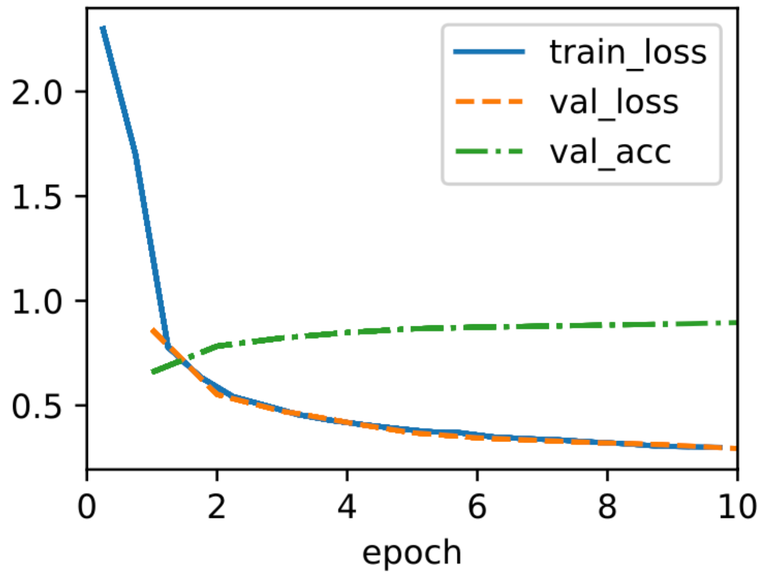
Networks Using Blocks (VGG)

```
VGG(arch=((1, 64),
          (1, 128),
          (2, 256),
          (2, 512),
          (2, 512))).layer_summary((1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:        torch.Size([1, 25088])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

Networks Using Blocks (VGG)

```
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



Network in Network (NiN)

Still Challenges:

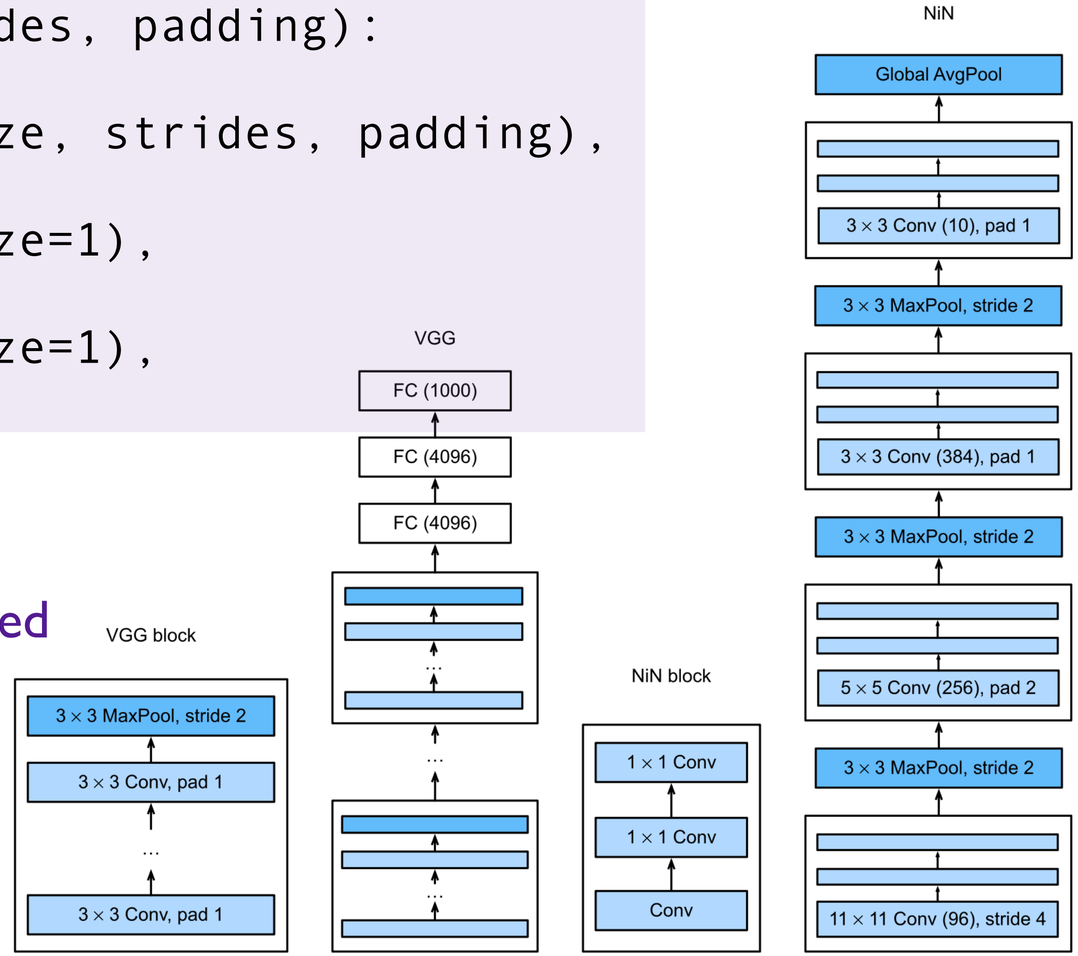
- **Parameter explosion in fully connected layers** — VGG-11 requires nearly 400 MB of RAM just for the final dense matrix, making it computationally heavy.
- **Hardware limitations** — such memory demand was impractical, especially for mobile/embedded devices with limited RAM (e.g., my iPhone 4S only had 512 MB of RAM).
- The *network in network* (*NiN*) blocks (Lin *et al.*, 2013) offer an alternative, capable of solving both problems in one simple strategy. Based on a very simple insight:
  - (i) use 1 × 1 convolutions to add local nonlinearities across the channel activations and
  - (ii) avoids FC layers altogether use global average pooling to integrate across all locations in the last representation layer.

Network in Network (NiN)

```
def nin_block(out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.LazyConv2d(out_channels, kernel_size, strides, padding),
        nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1),
        nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1),
        nn.ReLU())
```

Note: instead of FC layers ==> block with a number of output channels equal to the number of label classes, followed by a *global* average pooling layer, yielding a vector of logits.

This design significantly reduces the number of required model parameters, albeit at the expense of a potential increase in training time.



Network in Network (NiN)

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nin_block(96, kernel_size=11, strides=4, padding=0),
            nn.MaxPool2d(3, stride=2),
            nin_block(256, kernel_size=5, strides=1, padding=2),
            nn.MaxPool2d(3, stride=2),
            nin_block(384, kernel_size=3, strides=1, padding=1),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout(0.5),
            nin_block(num_classes, kernel_size=3, strides=1, padding=1),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten())
        self.net.apply(d2l.init_cnn)
```




Network in Network (NiN)

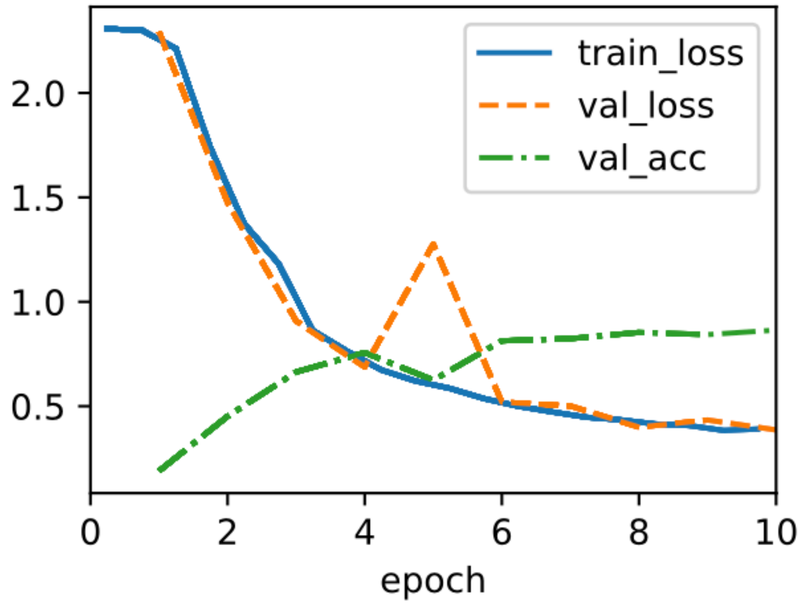
```
NiN().layer_summary((1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Sequential output shape:     torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Sequential output shape:     torch.Size([1, 384, 12, 12])
MaxPool2d output shape:      torch.Size([1, 384, 5, 5])
Dropout output shape:        torch.Size([1, 384, 5, 5])
Sequential output shape:     torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d output shape:      torch.Size([1, 10, 1, 1])
Flatten output shape:        torch.Size([1, 10])
```




Network in Network (NiN)

```
model = NiN(lr=0.05)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```

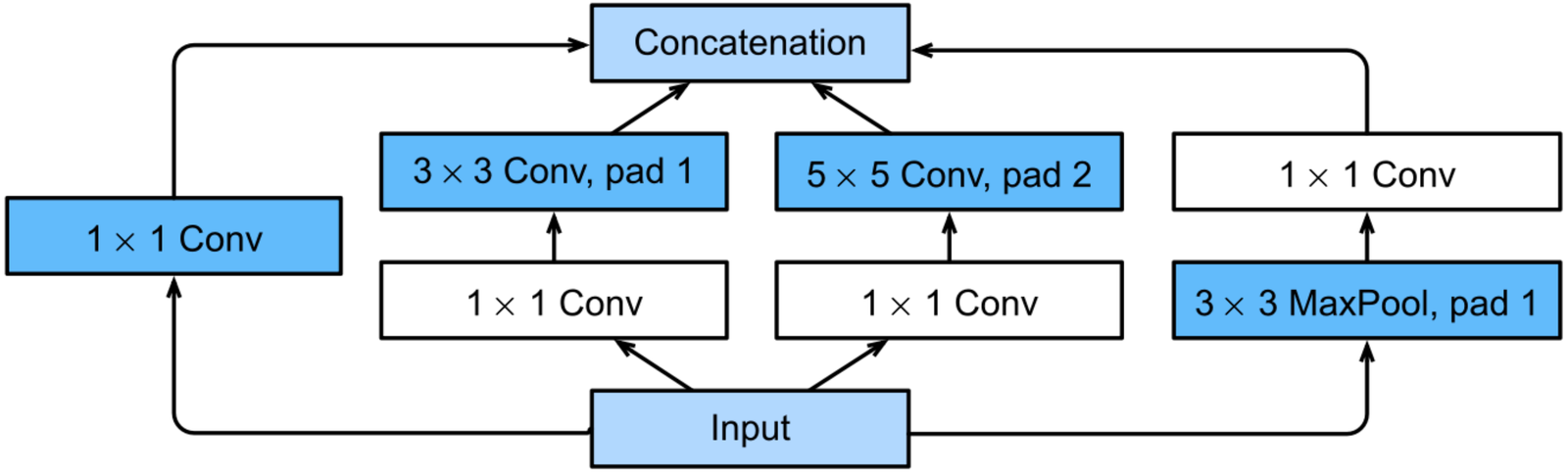


Multi-Branch Networks (GoogLeNet)

- **GoogLeNet's victory (2014, Szegedy et al., 2015)** — won ImageNet Challenge by combining ideas from NiN, VGG repeated blocks, and multiple convolution kernels.
- Clear CNN architecture pattern — introduced the now-standard split into **stem** (data ingest), **body** (processing), **head** (prediction).
- **Key innovation in the body** — instead of choosing one kernel size, GoogLeNet concatenated **multi-branch convolutions** (1×1, 3×3, 5×5, etc.).
- **Training stabilization tricks** — used auxiliary loss functions at intermediate layers, later made unnecessary by better training algorithms.

GoogLeNet — Inception Block

Called *Inception block*, from "we need to go deeper" from the movie *Inception*.



4 parallel branches:

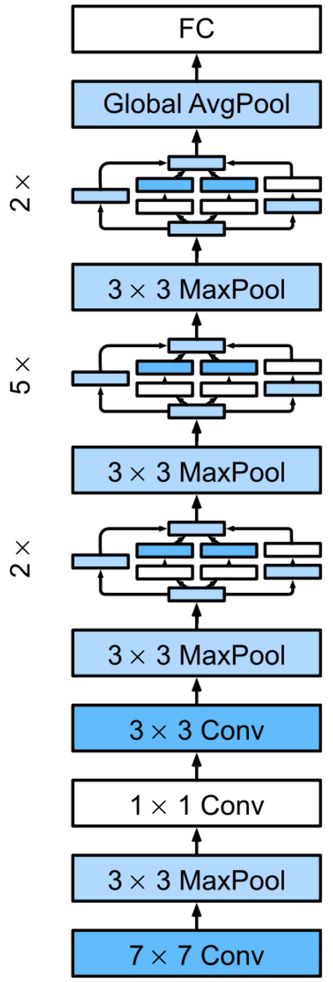
- The **first 3** branches use convolutional layers with window sizes of 1 × 1, 3 × 3, and 5 × 5 to extract information from different spatial sizes.
- The **middle 2** branches also add a 1 × 1 convolution of the input to reduce the number of channels, reducing the model's complexity.
- The **4th branch** uses a 3 × 3 max-pooling layer, followed by a 1 × 1 convolutional layer to change the number of channels.
- The four branches **all** use appropriate padding to give the input and output the same height and width.
- Finally, the **outputs** along each branch are concatenated along the channel dimension and comprise the block's output.

GoogLeNet — Incetoption Block

```
class Inception(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        # Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        # Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        # Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)
```

GoogLeNet

```
class GoogleNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
          nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
          nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    def b2(self):
      return nn.Sequential(
        nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
        nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    def b3(self):
       return nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
         Inception(128, (128, 192), (32, 96), 64),
         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    def b4(self):
      return nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
         Inception(160, (112, 224), (24, 64), 64),
         Inception(128, (128, 256), (24, 64), 64),
         Inception(112, (144, 288), (32, 64), 64),
         Inception(256, (160, 320), (32, 128), 128),
         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
    def b5(self):
      return nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
        Inception(384, (192, 384), (48, 128), 128),
        nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())
```



GoogLeNet

```
@d2l.add_to_class(GoogleNet)
def __init__(self, lr=0.1, num_classes=10):
    super(GoogleNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                             self.b5(), nn.LazyLinear(num_classes))
    self.net.apply(d2l.init_cnn)
```

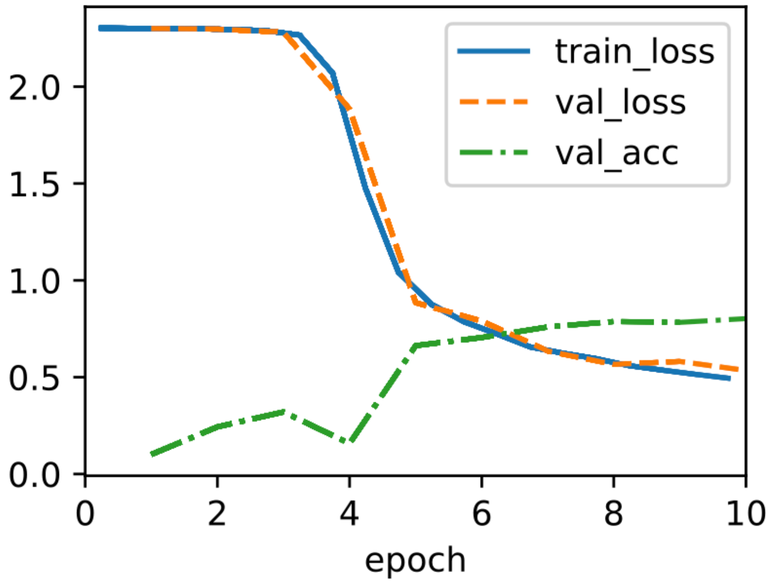
```
model = GoogleNet().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 192, 12, 12])
Sequential output shape:     torch.Size([1, 480, 6, 6])
Sequential output shape:     torch.Size([1, 832, 3, 3])
Sequential output shape:     torch.Size([1, 1024])
Linear output shape:         torch.Size([1, 10])
```



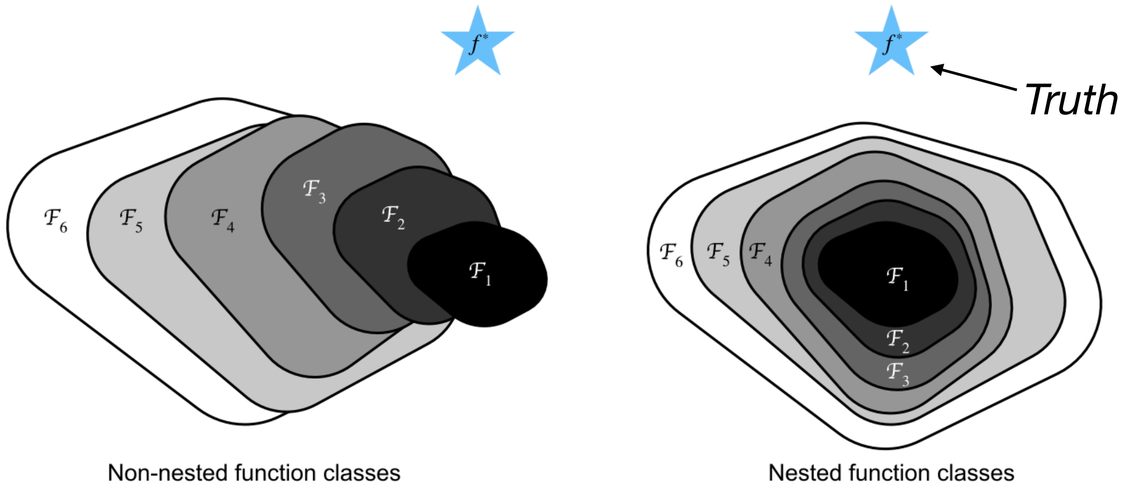

GoogLeNet

```
model = GoogleNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



Deeper NN are more difficult

- As we design ever deeper networks it becomes imperative to:
  - understand how adding layers can increase the complexity and expressiveness of the network.
  - Learn the ability to design networks where adding layers makes networks strictly more expressive rather than just different.



The best we can do training a NN is estimating a f_F^* that goes the closest to f^* among the possible functions given by sets of parameters of the NN we built.

There is no guaranteed that a different, more powerful architecture would perform better than a smaller one. **Only if larger function classes contain the smaller ones** we are guaranteed that increasing them strictly increases the expressive power of the network.

The ResNet Idea

- He *et al.* (2016) considered this problem when working on very deep computer vision models.

Problem: In principle, adding more layers to a neural network should allow it to model more complex functions. But in practice, deeper networks often perform *worse* — they're harder to optimize, and training error may actually go up when you add layers.

The ResNet idea

Suppose you have a trained model with some number of layers. Now imagine adding a new layer (or block of layers).

- If the new layer simply computes the **identity function**: f(x)=x, then the overall network behaves exactly like the original one — no harm done.
- This means: *at worst*, the deeper model can do at least as well as the shallower model (because it can always "skip" the new layer by acting like the identity).

The ResNet Idea

Why residual connections help

Instead of asking the new layers to directly learn a mapping H(x),
ResNets reframe the problem: the new layers learn a *residual* function

$$F(x) = H(x) - x$$

so that the output is

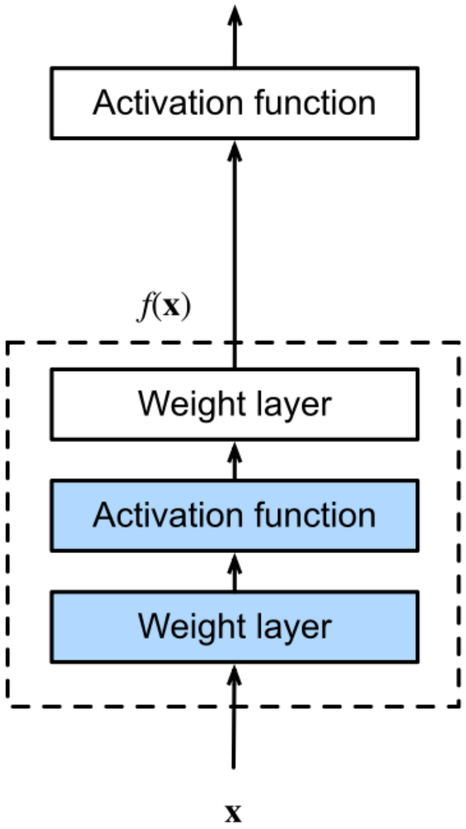
$$H(x) = F(x) + x$$

- If the residual F(x) is zero, the block just copies the input (identity).
- If learning a small adjustment is easier than learning the full mapping, optimization becomes much more stable.

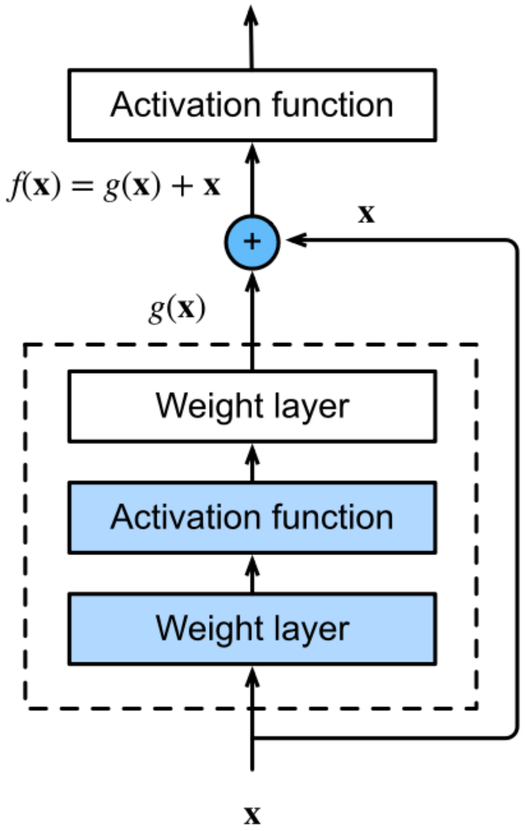
✴ ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015.

ResNet block

Regular block



Resnet block

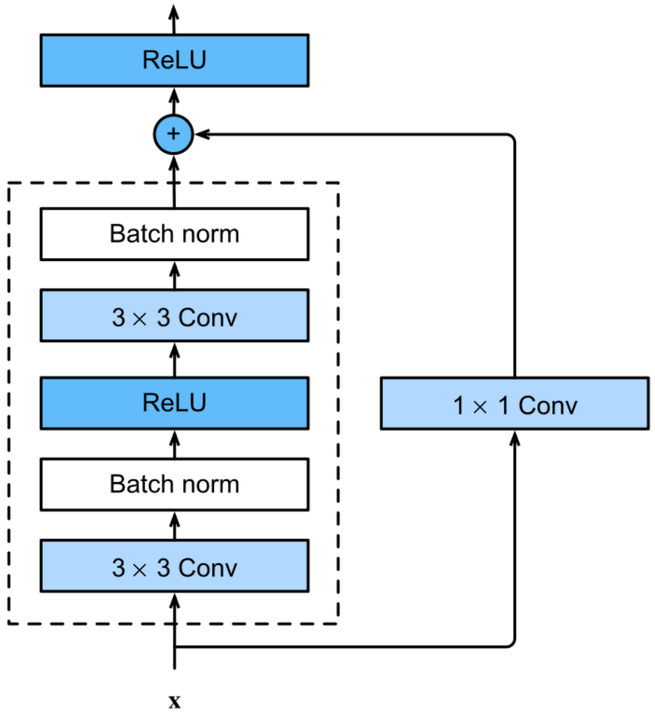


- Denote the input by x (this could be the output of a previous layer).
- We assume that $f(x)$, the mapping we want to obtain by learning, is to be used as input to the activation function on the top.
- **On the left,** the portion within the dotted-line box must directly learn $f(x)$.
- **On the right,** the portion within the dotted-line box needs to learn the *residual mapping* $g(x) = f(x) - x$.
  - If $f(x) = x$ is the desired underlying mapping, the residual $g(x) = 0$ and it is easier to learn: just push the weights and biases of the upper weight layer within the block to zero.

ResNet block

- ResNet has VGG's full 3 × 3 convolutional layer design.
- the output of the two convolutional layers has to be of the same shape as the input to be added
- If we want to change the number of channels, we need to introduce an additional 1 × 1 convolutional layer

```
class Residual(nn.Module):  #@save

    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=strides)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```



ResNet block

```
blk = Residual(3)
X = torch.randn(4, 3, 6, 6)
blk(X).shape
```

```
torch.Size([4, 3, 6, 6])
```

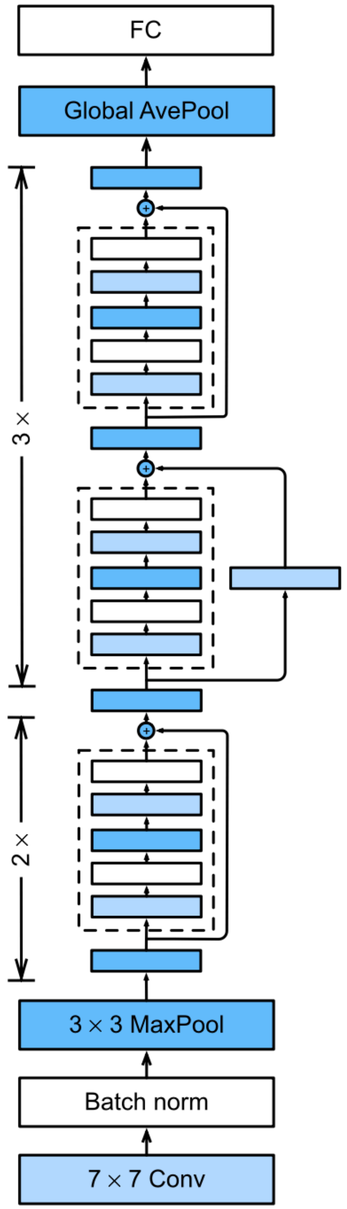
```
blk = Residual(6, use_1x1conv=True, strides=2) # conv1x1 convenient if stride>1
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```




ResNet

```
class ResNet(d2l.Classifier):

      def b1(self):
         return nn.Sequential(
              nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
              nn.LazyBatchNorm2d(), nn.ReLU(),
              nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

       def block(self, num_residuals, num_channels, first_block=False):
           blk = []
           for i in range(num_residuals):
               if i == 0 and not first_block:
                   blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
                 else:
                   blk.append(Residual(num_channels))
            return nn.Sequential(*blk)

       def __init__(self, arch, lr=0.1, num_classes=10):
           super(ResNet, self).__init__()
           self.save_hyperparameters()
           self.net = nn.Sequential(self.b1())
           for i, b in enumerate(arch):
               self.net.add_module(f'b{i+2}', self.block(*b, first_block=(i==0)))
           self.net.add_module('last', nn.Sequential(
               nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
               nn.LazyLinear(num_classes)))
           self.net.apply(d2l.init_cnn)
```



There are **4 conv** layers in **each module** (excluding the 1×1 convolutional layer). Together with the **first 7 × 7 conv** layer and the **final FC** layer, there are 18 layers in total.

Therefore, this model is commonly known as **ResNet-18**

ResNet-18

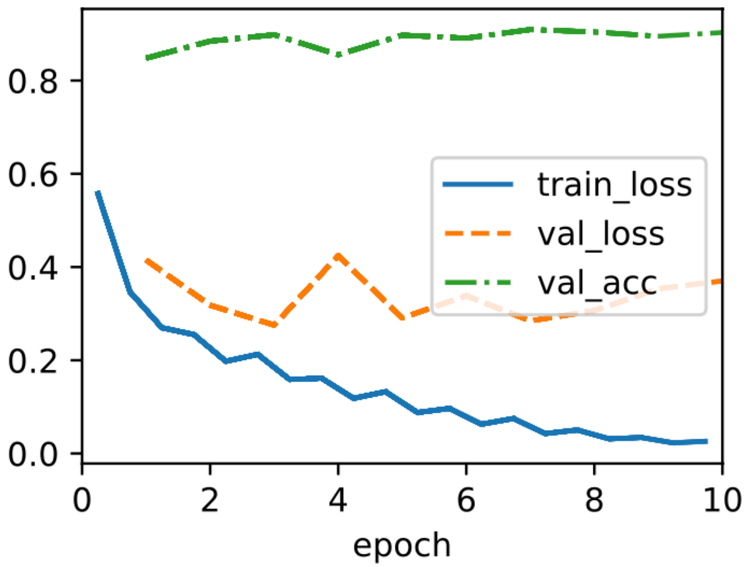
```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_classes)
ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 128, 12, 12])
Sequential output shape:     torch.Size([1, 256, 6, 6])
Sequential output shape:     torch.Size([1, 512, 3, 3])
Sequential output shape:     torch.Size([1, 10])
```



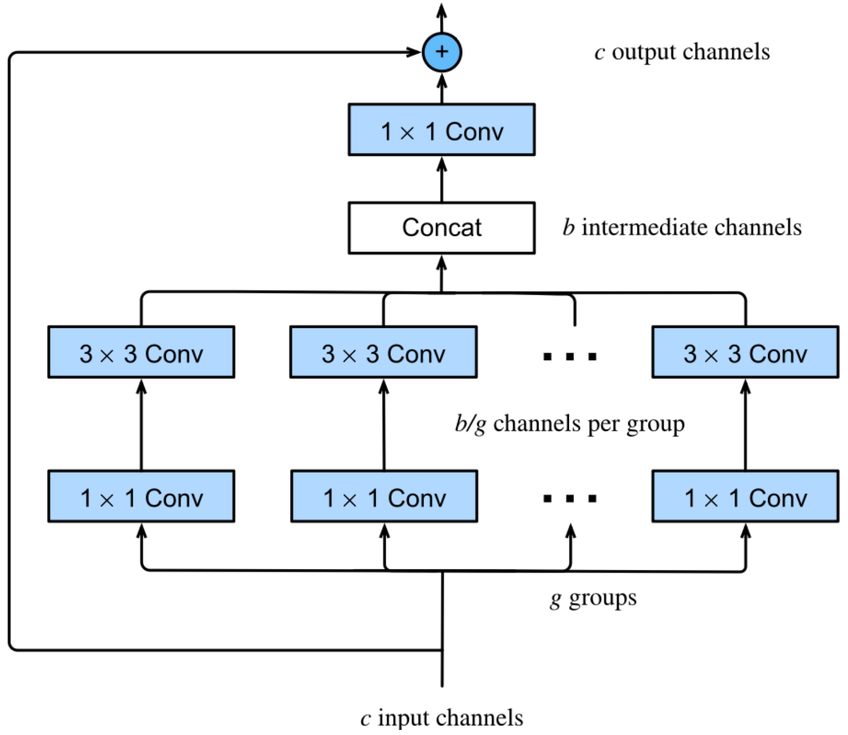

ResNet-18

```
model = ResNet18(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)
trainer.fit(model, data)
```



ResNext

- We can take some inspiration from the Inception block of Fig. 8.4.1 which has information flowing through the block in separate groups. Applying the idea of multiple independent groups to the ResNet block of Fig. 8.6.3 led to the design of ResNeXt (Xie *et al.*, 2017)

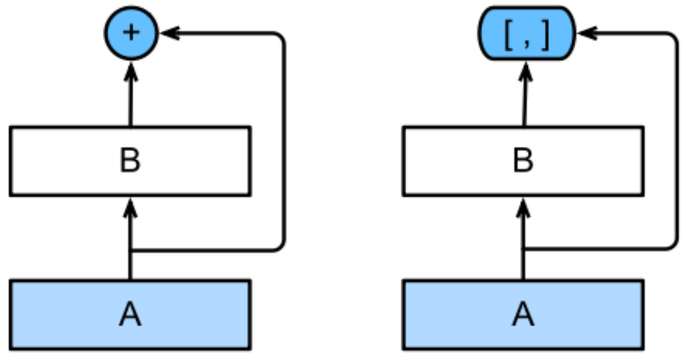


DenseNet

- ResNet significantly changed the view of how to parametrize the functions in deep networks.
- ***DenseNet*** (dense convolutional network) is to some extent the logical extension of this (Huang *et al.*, 2017), characterized by both the connectivity pattern where each layer connects to all the preceding layers and the concatenation operation (rather than the addition operator in ResNet) to preserve and reuse features from earlier layers.

Recall the Taylor expansion for functions: $$f(x) = f(0) + x \cdot \left[f'(0) + x \cdot \left[\frac{f''(0)}{2!} + x \cdot \left[\frac{f'''(0)}{3!} + \cdots \right] \right] \right]$$

The key point is that it decomposes a function into terms of increasingly higher order.
ResNet decomposes functions into $f(x) = x + g(x)$

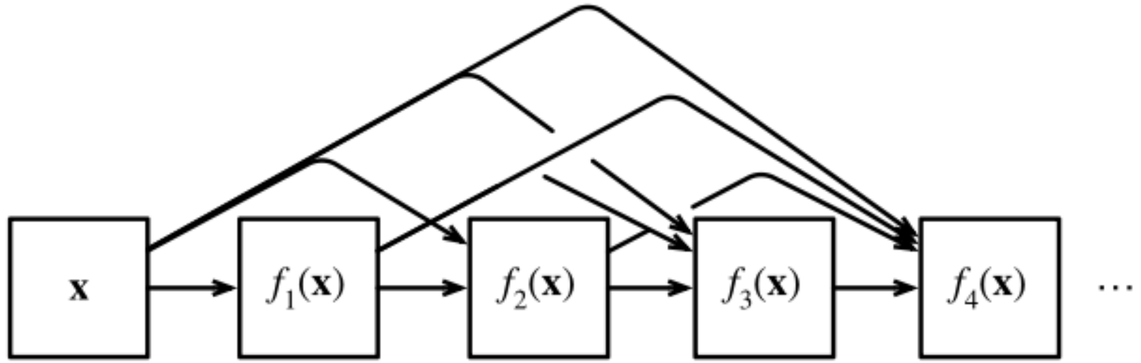


ResNet **DenseNet**

DenseNet

The key difference from ResNet: outputs are ***concatenated*** rather than added.
So we perform a mapping from x to its values after applying an increasingly complex sequence of functions:

$$\mathbf{x} \rightarrow [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots]$$



DenseNet

Two main components:

1. ***dense blocks:*** *define how the inputs and outputs are concatenated*
2. ***transition layers:*** control the number of channels so that it is not too large

DenseNet uses the "batch normalization, activation, and convolution" structure of ResNet

```
def conv_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1))
```




Dense block

dense block = multiple convolution blocks, each with the same number of channels.
In the forward propagation we concatenate the input and output of each convolution block on the channel dimension

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
       for blk in self.net:
            Y = blk(X)
            # Concatenate along the channels
            X = torch.cat((X, Y), dim=1)
       return X
```




Transition Block

```
def transition_block(num_channels):
    return nn.Sequential(
        nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A *transition layer* is used to control the complexity of the model. It reduces the number of channels by using a 1×1 convolution.




DenseNet

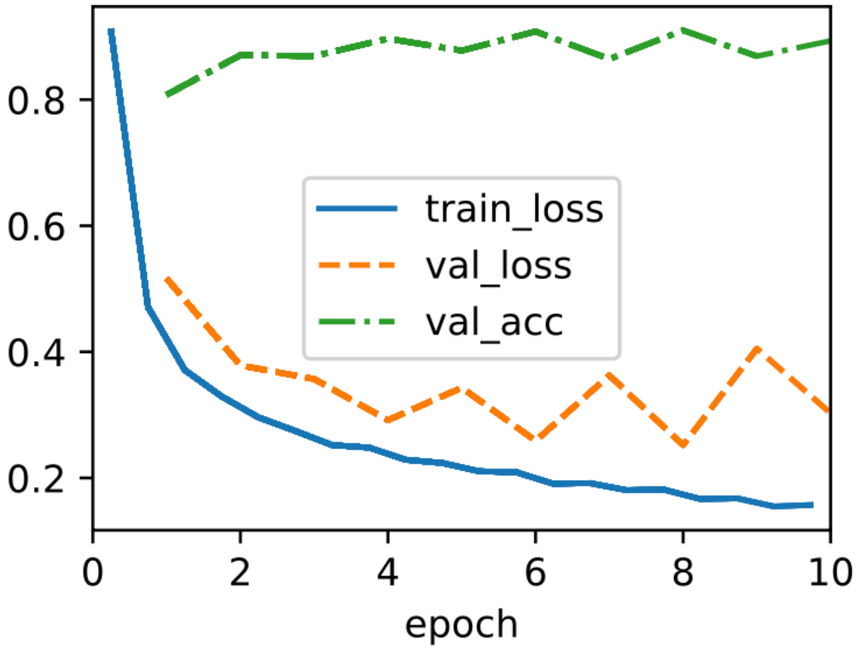
```
class DenseNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    def __init__(self, num_channels=64, growth_rate=32, arch=(4, 4, 4, 4),
                 lr=0.1, num_classes=10):
        super(DenseNet, self).__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(self.b1())
        for i, num_convs in enumerate(arch):
            self.net.add_module(f'dense_blk{i+1}', DenseBlock(num_convs,
                                                              growth_rate)
            if i != len(arch) - 1:
                num_channels //= 2
                self.net.add_module(f'tran_blk{i+1}', transition_block(
                    num_channels))
        self.net.add_module('last', nn.Sequential(
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes)))
        self.net.apply(d2l.init_cnn)
```

The number of output channels in the previous dense block: num_channels += num_convs * growth_rate.

A transition layer that halves the number of channels is added between the dense blocks

DenseNet

```
model = DenseNet(lr=0.01)
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
trainer.fit(model, data)
```

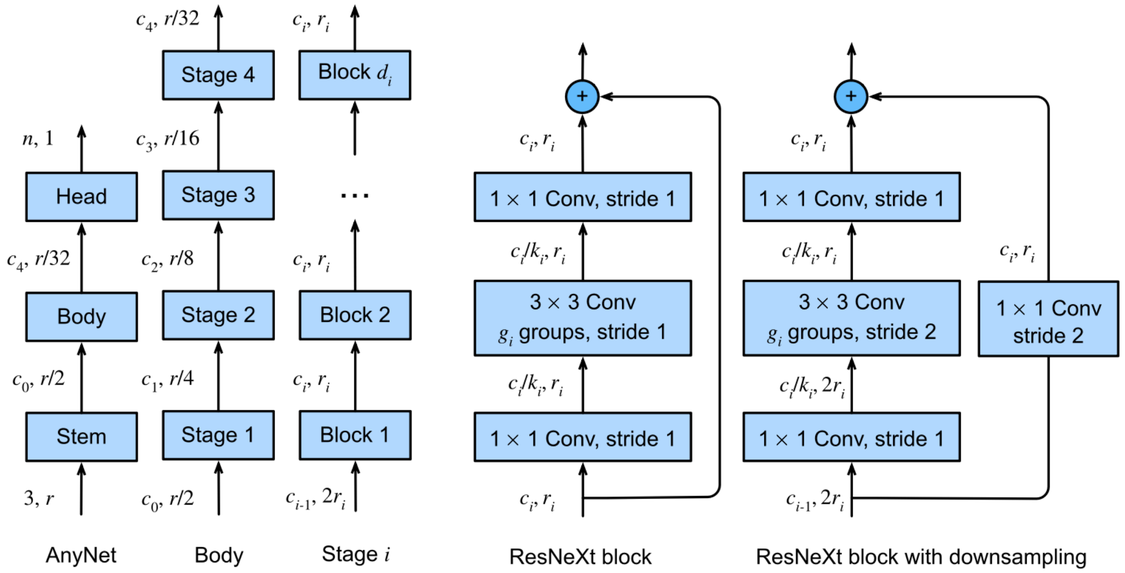


AnyNet

Radosavovic *et al.* (2020)

Stem: initial image processing, often using larger convolution windows
Body: composed of multiple blocks, performing most of the feature transformations
Head: converts features into outputs (e.g., softmax regressor for classification)

The **AnyNet** design space:
(c,r) = (# channels, resolution r × r) of the images at that point. From left to right: generic network structure composed of stem, body, and head; body composed of four stages; detailed structure of a stage; two alternative structures for blocks, one without downsampling and one that halves the resolution in each dimension. Design choices include depth di , the number of output channels ci , the number of groups gi , and bottleneck ratio ki for any stage i.

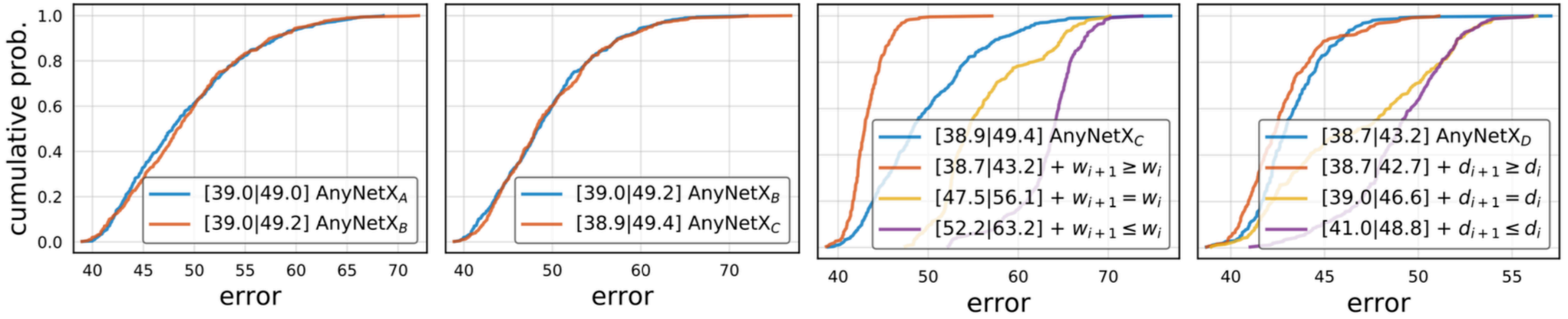


https://d2l.ai/chapter_convolutional-modern/cnn-design.html

AnyNet

- Strategy to determine general guidelines of how the choices of parameters should be related: (**hyper-parameters and structure optimization**)
  - We assume that general design principles actually exist
  - We need not train networks to convergence before we can assess whether a network is good.
  - Results obtained at a smaller scale (for smaller networks) generalize to larger ones
  - Aspects of the design can be approximately factorized (infer their effect on the quality of the outcome somewhat independently)

Design optimization



AnyNetA is the original design space; AnyNetB ties the bottleneck ratios, AnyNetC also ties group widths, AnyNetD increases the network depth across stages.

From left to right:

(i) tying bottleneck ratios has no effect on performance;
(ii) tying group widths has no effect on performance;
(iii) increasing network **widths** (channels) across stages improves performance;
(iv) increasing network **depths** across stages improves performance. Radosavovic et al. (2020)